

# 苏州大学实验报告

院、系	计算机学院	年级专业	软件工程	姓名	朱金涛	学号	2327406014
课程名称	操作系统课程实践					成绩	
指导教师	王红玲	同组实验者	无	实验日期	2025 年 9 月 15 日		

实验名称Linux 进程通信

## 一、实验目的

初步了解 Linux 系统中，创建进程和进程间通信的方法。

## 二、实验内容

- 实验 Linux 下创建子进程及资源共享的方法。
- 编写一个程序，用 Linux 中的 IPC 机制，完成两个进程“石头、剪子、布”的游戏。

## 三、实验步骤和结果

### 1. 【数据结构定义】

通过宏定义的方式分别令石头为 0、剪刀为 1、布为 2，后续在裁判进程中界定好判断规则：

```
// 判断胜负：石头>剪刀>布>石头
int judge_winner(int move1, int move2) {
    if (move1 == move2) return 0; // 平局
    if ((move1 == ROCK && move2 == SCISSORS) ||
        (move1 == SCISSORS && move2 == PAPER) ||
        (move1 == PAPER && move2 == ROCK)) {
        return 1; // move1胜
    }
    return -1; // move1负
}
```

同时定义好比拼的总轮数，本次实验设为 100。

## 2. 【结果存放】

程序中定义结构体，用于存放一轮中两个选手的出招：player1\_move、player2\_move 以及本轮对决的结果：result1、result2。当前轮数：round。选手当前总得分：player1\_score、player2\_score。游戏结束标志：game\_over。

此外，每轮结束后会将该轮结果存放于 txt 文件当中，效果图如下：

- 1 第1回合：剪刀 vs 布 - 选手1胜
- 2 第2回合：布 vs 剪刀 - 选手2胜
- 3 第3回合：布 vs 布 - 平局
- 4 第4回合：石头 vs 石头 - 平局
- 5 第5回合：石头 vs 剪刀 - 选手1胜
- 6 第6回合：布 vs 剪刀 - 选手2胜
- 7 第7回合：石头 vs 布 - 选手2胜
- 8 第8回合：布 vs 剪刀 - 选手2胜
- 9 第9回合：布 vs 石头 - 选手1胜
- 10 第10回合：剪刀 vs 布 - 选手1胜
- 11 第11回合：布 vs 剪刀 - 选手2胜

## 3. 【IPC】本实验选用共享内存+信号量的方法实现通信。

**理由：**共享内存作为高效的进程间数据交换载体，通过让多进程直接访问同一块物理内存，彻底避免了数据在用户空间与内核空间的多次拷贝，能极大提升高频、大量数据共享的效率，且可直接读写复杂数据结构，简化共享逻辑；而信号量则是保障共享资源安全访问的核心工具，其原子性的 P/V 操作能有效解决多进程间的同步与互斥问题，防止竞争条件导致的数据错乱，支持进程按特定逻辑协作；两者结合时，共享内存提供高效数据

传输能力，信号量提供安全访问控制，可在多进程场景中实现效率与数据一致。

资源创建：

```
// 信号量操作（对信号集中的第0个元素操作，执行P/V操作，默认模式）
struct sembuf sem_lock = {0, -1, 0};
struct sembuf sem_unlock = {0, 1, 0};

// 创建共享内存
int shmid = shmget(key, sizeof(GameData), IPC_CREAT |
0666);
if (shmid == -1) {
    perror("shmget");
    exit(1);
}

// 创建信号量
semid = semget(key, 1, IPC_CREAT | 0666);
if (semid == -1) {
    perror("semget");
    exit(1);
}
```

连接共享内存：

```
// 连接共享内存
game_data = (GameData*)shmat(shmid, NULL, 0);
if (game_data == (void*)-1) {
    perror("shmat");
    exit(1);
}
```

#### 4. 【选手进程创建】

选手逻辑实现：player\_process 函数定义了选手进程的行为：

(1) 随机生成出拳 (rand() % 3) 。

① 通过信号量 (semop) 控制共享内存的访问，确保安全读写自己的出拳 (player1\_move 或 player2\_move) 。

② 循环检查 game\_over 标志，结束后退出进程。

```
1 // player_process函数中
2 while (!game_data->game_over) {
3     semop(semid, &sem_lock, 1); // P操作
4
5     // 临界区开始：只有抢到锁的进程能运行到这
6     if (game_data->round > 0) {
7         // 检查自己是不是还没出招 (值为-1表示没出招)
8         if (player_id == 1 && game_data->player1_move == -1) {
9             game_data->player1_move = rand() % 3; // 写共享内存
10        }
11        // ... player2同理
12    }
13    // 临界区结束
14
15    semop(semid, &sem_unlock, 1); // V操作
16    usleep(10000);
17 }
```

(2) 进程创建：main 函数中通过 fork() 创建两个子进程，分别调用 player\_process(1) 和 player\_process(2) 作为选手 1 和选手 2。

#### 5. 【裁判进程创建】

(1) 裁判逻辑实现：judge\_process 函数定义了裁判进程的行为：

- ① 控制回合数（1 到 ROUNDS），初始化每回合的出拳状态。
- ② 等待两位选手完成出拳后，调用 `judge_winner` 判断胜负并更新分数。
- ③ 输出每回合结果到控制台和文件，最终统计并输出总结果。

```
1 // judge_process函数 - 等待逻辑
2 while (1) {
3     semop(semid, &sem_lock, 1); // 加锁
4
5     // 检查共享内存
6     if (game_data->player1_move != -1 && game_data->player2_move != -1) {
7         semop(semid, &sem_unlock, 1); // 都出手了，解锁并跳出循环
8         break;
9     }
10
11     semop(semid, &sem_unlock, 1); // 还没齐
12     usleep(1000); // 等一毫秒再次检查，直到都出手
13 }
```

- (2) 进程运行：主进程（未被 `fork()` 复制的原始进程）直接调用 `judge_process()` 作为裁判进程。

## 6. 【运行结果】

```
● anders@anders-Legion-Y9000P-IRX8:~/OS_Operation/op003$ ./001
石头剪刀布比赛开始！共100回合
第1回合：剪刀 vs 布 - 选手1胜
第2回合：布 vs 剪刀 - 选手2胜
第3回合：布 vs 布 - 平局
第4回合：石头 vs 石头 - 平局
第5回合：石头 vs 剪刀 - 选手1胜
第6回合：布 vs 剪刀 - 选手2胜
第7回合：石头 vs 布 - 选手2胜
第8回合：布 vs 剪刀 - 选手2胜
第9回合：布 vs 石头 - 选手1胜
第10回合：剪刀 vs 布 - 选手1胜
第11回合：布 vs 剪刀 - 选手2胜
第12回合：剪刀 vs 石头 - 选手2胜
第13回合：剪刀 vs 石头 - 选手2胜
第14回合：剪刀 vs 石头 - 选手2胜
第15回合：石头 vs 石头 - 平局
第16回合：剪刀 vs 石头 - 选手2胜
第17回合：布 vs 剪刀 - 选手2胜
第18回合：石头 vs 剪刀 - 选手1胜
第19回合：布 vs 石头 - 选手1胜
第20回合：石头 vs 石头 - 平局
```

```
第90回合：石头 vs 石头 - 平局
第91回合：石头 vs 布 - 选手2胜
第92回合：布 vs 石头 - 选手1胜
第93回合：剪刀 vs 剪刀 - 平局
第94回合：石头 vs 石头 - 平局
第95回合：剪刀 vs 石头 - 选手2胜
第96回合：布 vs 布 - 平局
第97回合：剪刀 vs 布 - 选手1胜
第98回合：石头 vs 剪刀 - 选手1胜
第99回合：布 vs 布 - 平局
第100回合：石头 vs 剪刀 - 选手1胜
```

```
=== 比赛结果 ===
选手1得分：38
选手2得分：32
平局数：30
选手1获胜！
游戏结束，结果已保存到 game_results.txt
```

## 7. 【选做】

决出年级前三：

```
问题 输出 调试控制台 终端 端口
./class_championship
第19回合：石头 vs 布 - 二班选手3胜
第20回合：石头 vs 石头 - 平局
比赛结果：8 - 5

=== 年级总排名 ===
1. 二班选手6 - 总分:56 胜率:0.62 🏆 年级冠军 🏆
2. 一班选手6 - 总分:56 胜率:0.38 🥈 年级亚军 🥈
3. 二班选手3 - 总分:50 胜率:0.12 🥉 年级季军 🥉
4. 一班选手5 - 总分:39 胜率:0.60
5. 二班选手1 - 总分:37 胜率:0.80
6. 一班选手2 - 总分:36 胜率:0.40

🎉 比赛圆满结束！详细结果已保存到 championship_results.txt
anders@anders-Legion-Y9000P-IRX8:~/OS_Operation/op003$
```

通过共享内存实现数据共享、信号量保证进程同步，以多进程模拟“石头剪刀布”锦标赛，整体逻辑分为初始化、进程创建、赛事执行和资源清理四部分：首先创建共享内存存储选手信息、比赛状态等数据，并用信号量保护共享资源的访问；接着创建 12 个选手进程（每班 6 名）和 1 个主裁判进程，选手进程循环等待比赛信号并随机出拳，主裁判进程则负责赛事统筹——先初始化选手数据，依次组织一班、二班的内部循环赛（每位选手与同班级其他选手对战 20 回合，按总分和胜率排序选出各班前三甲），再安排两班

前三甲交叉进行年级赛，最后计算年级总排名并将结果写入日志文件；赛事结束后，主裁判标记比赛结束，等待所有选手进程退出，再清理共享内存和信号量资源。

### 网络版：

编写 `sever.c` 和 `client.c`，分别实现服务端和客户端。

#### ➤ 服务端的核心代码实现：

一直死循环等待，一旦有新连接，就会创建一个新的线程去服务该客户端。

```
1 while (1) {
2     // 1. 阻塞等待新连接
3     client_fd = accept(server_fd, ...);
4
5     // ... 查找空闲位置 ...
6
7     // 2. 为这个客户端创建一个独立的线程
8     // 这样主线程可以立刻回去 accept 下一个人，不会被当前玩家卡住
9     pthread_create(&clients[client_index].thread, NULL, handle_client, &clients[client_index]);
10 }
```

创建的一个房间中具体的判断逻辑：

```
1 void handle_player_move(Room *room, int player_id, int move) {
2     pthread_mutex_lock(&room->mutex); // 【加锁】防止两个人同时写
3
4     // 1. 记录当前玩家的出招
5     if (player_id == room->player1_id) room->player1_move = move;
6     else if (player_id == room->player2_id) room->player2_move = move;
7
8     // 2. 两个人是否都出招了？
9     if (room->player1_move != -1 && room->player2_move != -1) {
10
11         // 3. 判胜负
12         int result = judge_winner(room->player1_move, room->player2_move);
13
14         // 4. 广播结果给房间里所有人
15         broadcast_to_room(room->id, &msg);
16     }
17     pthread_mutex_unlock(&room->mutex); // 【解锁】
18 }
```

#### ➤ 双线程结构实现客户端

之所以用双线程，是因为客户端既要等待用户输入，又要随时接收服务器的消息（聊天、对手出招结果）。如果只用一个线程，程序会在等待用户输入时卡死，收不到服务器发来的“对手已出招”或“聊天消息”。

专门负责监听服务端消息的线程：

```
1 void* receive_messages(void* arg) {
2     while (game_running) {
3         // 阻塞等待服务器发消息
4         int received = receive_message(client_socket, &msg);
5
6         // 收到消息后，根据类型直接打印到屏幕
7         switch (msg.type) {
8             case MSG_CHAT: printf("💬 %s\n", msg.data); break;
9             case MSG_RESULT: printf("🎯 %s\n", msg.data); break;
10            // ...
11        }
12    }
13 }
```

负责与用户交互的线程：

```
1 while (game_running) {
2     fgets(input, ...); // 等待用户打字
3
4     if (client_state == CLIENT_PLAYING) {
5         // 如果是游戏中，输入的数字就是出招
6         msg.type = MSG_MOVE;
7         send_message(client_socket, &msg);
8
9     } else if (client_state == CLIENT_IN_ROOM) {
10        // 如果在房间里没玩游戏，输入的就是聊天内容
11        msg.type = MSG_CHAT;
12        send_message(client_socket, &msg);
13    }
14 }
```

功能展示：

客户端和服务端的连接：

```
问题 输出 调试控制台 终端 窗口
e run-client
提示：
• 在房间中可以聊天
• 输入 /help 查看帮助
• 输入 /leave 离开房间
• 按 Ctrl+C 退出程序

===== 石头剪刀布网络版 =====
1. 查看房间列表
2. 创建房间
3. 加入房间
4. 帮助
5. 退出
请选择 (1-5): 1

e run-server
./server
石头剪刀布服务器启动成功！
监听端口: 8888
最大客户端数: 10
最大房间数: 5
客户端 玩家1 已连接 (IP: 127.0.0.1)
客户端 玩家2 已连接 (IP: 127.0.0.1)
```

创建房间：

```
问题 输出 调试控制台 终端 窗口
e run-client
===== 石头剪刀布网络版 =====
1. 查看房间列表
2. 创建房间
3. 加入房间
4. 帮助
5. 退出
请选择 (1-5): 2
请输入房间名称: zjt
成功创建房间 'zjt', 等待其他玩家加入 ...
第1回合开始! 请出招: 0=石头 1=剪刀 2=布

e run-server
./server
石头剪刀布服务器启动成功！
监听端口: 8888
最大客户端数: 10
最大房间数: 5
客户端 玩家1 已连接 (IP: 127.0.0.1)
客户端 玩家2 已连接 (IP: 127.0.0.1)
```

出招：

```
问题 输出 调试控制台 终端 窗口
e run-client
4. 帮助
5. 退出
请选择 (1-5): 3
请输入房间号: zjt
成功加入房间0, 游戏即将开始!
第1回合开始! 请出招: 0=石头 1=剪刀 2=布
第1回合: 玩家2(剪刀) vs 玩家1(布) - 玩家2胜!
第2回合开始! 请出招: 0=石头 1=剪刀 2=布

e run-server
./server
石头剪刀布服务器启动成功！
监听端口: 8888
最大客户端数: 10
最大房间数: 5
客户端 玩家1 已连接 (IP: 127.0.0.1)
客户端 玩家2 已连接 (IP: 127.0.0.1)
```

测试平局结果是否可行：

```
问题 输出 调试控制台 终端 窗口
e run-client
请输入房间名称: zjt
成功创建房间 'zjt', 等待其他玩家加入 ...
第1回合开始! 请出招: 0=石头 1=剪刀 2=布
第1回合: 玩家2(剪刀) vs 玩家1(布) - 玩家2胜!
第2回合开始! 请出招: 0=石头 1=剪刀 2=布
第2回合: 玩家2(布) vs 玩家1(布) - 平局!
第3回合开始! 请出招: 0=石头 1=剪刀 2=布

e run-server
./server
石头剪刀布服务器启动成功！
监听端口: 8888
最大客户端数: 10
最大房间数: 5
客户端 玩家1 已连接 (IP: 127.0.0.1)
客户端 玩家2 已连接 (IP: 127.0.0.1)
```

基于 TCP 网络通信和多线程实现的多人在线“石头剪刀布”游戏系统，分客户端与服务器端协同工作：服务器端初始化后监听 8888 端口，支持最多 10 个客户端连接和 5 个房间，通过多线程为每个客户端提供独立服务，用互斥锁保护共享的客户端 / 房间数据，可处理创建房间、加入房间、查看房间列表等请求，当房间凑齐 2 人后启动 10 回合对战，自动裁决每回合胜负并统计比分，还支持房间内聊天广播；客户端负责连接服务器，通过独立接收线程实时获取服务器消息（状态提示、对战结果、聊天内容等），根据自身状态（大厅 / 房间内 / 游戏中）响应用户操作——大厅可选择菜单功能，房间内可聊天或执行命令，游戏中可输入 0-2 出招，整体通过统一的消息结构和类型约定实现两端数据交互与逻辑同步。

## 四、 实验总结

本次 Linux 进程通信实验以“石头、剪子、布”游戏为载体，成功实现了多进程的创建与协同通信，达成了初步掌握 Linux 系统进程创建及进程间通信方法的实验目的。通过实践，对共享内存与信号量结合的 IPC 机制有了直观且深入的理解，验证了该机制在多进程数据交互中的高效性与安全性。

在实验设计与实现层面，我们首先通过宏定义明确了“石头、剪刀、布”的数值映射及胜负判断逻辑，并借助结构体规范了游戏数据的存储格式，涵盖选手出招、回合结果、得分统计等关键信息，为进程间数据共享奠定了清晰的数据基础。同时，设定 100 轮对决的实验场景，确保了对通信机制稳定性的充分测试。

进程通信方案的选择是本次实验的核心。我们最终采用共享内存搭配信号量的组合方式，这一选择既利用了共享内存直接访问物理内存、避免数据多次拷贝的高效优势，实现了复杂游戏数据结构的直接读写；又通过信号量的原子性 P/V 操作，有效解决了多进程访问共享资源时的同步与互斥问题，防止了竞争条件导致的数据错乱。在资源创建与连接环节，通过 `semget` 创建信号量、`shmat` 连接共享内存的操作均顺利完成，为进程通信提供了可靠的基础设施。

进程创建与逻辑实现环节同样达到了预期效果。主进程通过 `fork()` 函数成功创建两个子进程作为参赛选手，子进程能够通过随机数生成实现自主出招，并在信号量的控制下安全读写共享内存中的出招数据；主进程直接承担裁判进程职责，按照回合顺序完成选手出招等待、胜负判定、分数更新、结果输出及文件写入等一系列逻辑，整个进程间的协作流程清晰且有序。

从运行结果来看，100 轮对决的过程数据实时输出至控制台，最终统计显示选手 1 得分

38、选手 2 得分 32、平局 30 次, 选手 1 获胜, 且所有回合结果均成功保存至 game\_results.txt 文件中, 数据完整且准确。此外三种情况接近 1: 1: 1, 符合实际概率。这表明实验所实现的进程创建、通信及协同逻辑均运行正常, 达到了实验设计的预期目标。

此次实验也暴露了部分可优化空间, 例如选手进程等待出招的循环检查方式可能存在一定的资源消耗, 未来可考虑通过信号量的更精细设计进一步提升进程协作效率。但总体而言, 本次实验成功将 Linux 进程通信理论与实际应用相结合, 不仅扎实掌握了 fork()进程创建、共享内存与信号量的使用方法, 更深刻理解了多进程同步与互斥的核心原理, 为后续更复杂的 Linux 系统编程实践积累了宝贵经验。